Christopher Kon

CS 320

Professor Martinez

4/20/2025

Project Two

During Project One, my unit testing approach aligned JUnit test cases with the customer’s functional requirements for the Contact, Task, and Appointment services. Each service class was built to manage in-memory data, and my testing strategy reflected this lightweight yet precise scope. For the Contact class, my approach ensured that all fields adhered to specific constraints. The contact ID, for example, could not exceed 10 characters, be null, or be modified after instantiation. I validated these requirements by writing JUnit tests, such as testContactIdTooLong() and testContactIdNull(), which asserted that invalid IDs resulted in exceptions. Similarly, for the Task service, tests like testUpdateTaskName() confirmed that updates behaved correctly, while testDeleteTask() ensured that deleted tasks were no longer accessible from the internal list.

My approach remained directly aligned with the software requirements throughout the testing process. Each functional requirement from the project prompt was traced to one or more test cases. For example, the Appointment class required a unique ID and a future date. I validated these by implementing testAppointmentDateInPast() to throw an exception when the appointment time was set to a past date, aligning the test case with real-world logic and system expectations. The adherence to specifications helped reduce the risk of requirement misinterpretation.

I am confident in the overall quality of my JUnit tests due to the completeness of test coverage and robustness of edge case analysis. I used Eclipse's code coverage plugin to verify that each method and constructor in my source classes was exercised by at least one test. For example, in AppointmentServiceTest, the method testAddAppointment() verified that the object was correctly added by asserting a non-null return and comparing the stored ID with the input value. assertNotNull(service.getAppointment("APT001")); assertEquals("APT001", service.getAppointment("APT001").getAppointmentId());

I minimized the likelihood of bugs slipping through undetected by ensuring that both typical and boundary cases were tested. Writing the JUnit tests was a technical yet insightful experience. I relied on assertions such as assertThrows, assertEquals, and assertTrue to verify method behavior. One example of this is in TaskTest.java, where I used the following line to ensure proper updating of a task description: assertEquals("Updated Description", task.getDescription()). I avoided unnecessary object instantiations to keep the tests efficient and reused service instances when possible. Furthermore, each test method was short and specific to a single behavior, allowing easier debugging and isolating issues.

The testing techniques I employed during this project primarily included unit testing, black-box testing, and boundary value analysis. Unit testing allowed me to isolate specific methods in the Contact, Task, and Appointment services and validate their functionality independently. Black-box testing focused on input/output behavior without internal code structure knowledge, ensuring that I tested each method from the perspective of an external user. For example, I did not rely on how the AppointmentService internally stored data but instead checked whether retrieval and deletion worked correctly from the outside. Boundary value analysis helped test extreme input cases such as minimum and maximum character limits for contact IDs and task descriptions.

While I did not use integration or system testing for this project, these techniques are worth noting. Integration testing focuses on interactions between different components, such as how a database layer might communicate with a service layer. This technique would be valuable in a more complex application where services depend on one another or external APIs. System testing evaluates the end-to-end functionality of the entire system in a production-like environment. For example, it would test how a UI interacts with the backend to create a new appointment or task.

Each of these techniques has practical implications. Unit testing is essential for code correctness during development, especially for regression prevention. Integration testing becomes critical when multiple modules interact, and system testing is key before deployment to ensure the software functions as a whole. Depending on the project's stage of development and complexity, these techniques can be layered to build confidence in the software’s reliability.

Throughout the project, I adopted a cautious and analytical mindset. As a tester, I treated each method as potentially faulty and approached testing skeptically. This mindset was crucial because even simple logic could have hidden issues. For example, while testing the deleteContact() method, I validated that the deleted ID was no longer accessible, rather than assuming the remove() method would work as expected. This approach encouraged critical thinking about the code's assumptions and helped catch subtle mistakes.

Limiting bias was another focus. Since I wrote both the implementation and the tests, I recognized the potential for unintentional confirmation bias testing for what I expected to work rather than what could go wrong. To counter this, I challenged my code by intentionally passing invalid inputs. For instance, I passed null values and strings exceeding character limits to ensure the software would handle them correctly. I might have missed critical failure points if I had only tested with ideal inputs. Additionally, I occasionally stepped away from the code and reviewed it later with a fresh perspective, which helped me identify areas for improvement.

Maintaining a commitment to quality was essential. It can be tempting to cut corners by skipping validation or assuming specific methods will work, especially under tight deadlines. However, this leads to technical debt shortcuts that may cause more significant problems in the future. I practiced defensive programming to avoid technical debt and enforced strict validation rules in my services. For example, I added explicit exception handling in constructors and setters to prevent invalid state creation. In the future, I plan to integrate tools like static code analyzers and continuous integration systems (e.g., GitHub Actions) into my workflow to enforce quality standards and automatically flag issues before they reach production.

Project One served as a valuable experience in developing and testing software using a structured, requirement-based approach. I demonstrated that effective unit testing ensures correctness and provides a framework for validating software quality throughout development. I ensured that the Contact, Task, and Appointment services met customer expectations by applying multiple testing techniques, adopting a critical and unbiased mindset, and staying disciplined about quality. These principles will guide my future practice as a software engineer and reinforce the importance of building robust, maintainable software systems.

References

Gamma, E., Beck, K., & Kent, R. (1999). JUnit: A cookbook approach to unit testing. Addison-Wesley.JUnit 5 Team. (n.d.). User Guide. https://junit.org/junit5/docs/current/user-guide/